```
//------------------------------------------------------------------------------------------------------------------
// GENERAL NOTES
//------------------------------------------------------------------------------------------------------------------

//-------------------------------------------
// LIBRARIES
//-------------------------------------------
        #include<iostream>        // it has also: std::max

        // Sequential containers
        #include<vector>          // std::vector (put "using std::vector;")
        #include<string>
        #include<queue>
        #include<deque>           // deques are double-ended queues with faster access
        #include<list>
        #include<forward_list>

        // Associative containers
        #include<map>             // std::map (put "using std::map;")
        #include<unordered>map>   // std::unordered_map (put "using std::unordered_map;")
        #include<utility>         // std::pair (put "using std::pair")

        // Shared pointers
        #include<memory>

        // MPI
        #include<mpi.h>

        // Addictional libraries
        #include<math.h>          // (no std::) sqrt(..), atan(..), cos(..), sin(..)
        #include<cmath>           // std::abs(..)
        #include<random>          // to generate random numbers (see: "RANDOM")
        #include<limits>          // std::numeric_limits<double>::quit_NaN(); (see: "RANDOM")
        #include<functional>

//-------------------------------------------
// CLASSES
//-------------------------------------------
// * In const method of a class:
//      - we CANNOT modify non-static members
//      - we CANNOT call other non-static method which are NOT const
//      - we CANNOT return a plain reference to a non-static data member

// * If two classes depends on each other we need a first declaration:
        class Newton;
        class Apple{
            void hit__on__head(Newton&);
        };
        class Newton{
            void bite(Apple&);
        };

// * Getters are const!

// * HEADER structure
        #ifndef CLASSNAME_H
        #define CLASSNAME_H
        #include<..>         // e.g. #include<vector>
        #include ".."        // e.g. #include "OtherClass.h"
        class ClassName{ .. };
        #endif // CLASSNAME_H

// * (!) CONST: we always have to think "is this method going to change the state?" -> if no, CONST.

// * When we're dealing with references and const methods -> we must pass CONST references!

// * When we return and obj of the class, we can create directly on the return:
        return Example(0);   // calling the costructor of Example with init 0

// * Function "add_new_element(..)"? First in the function: check if the element is already there.

// * Polymorphis: we can override multiple times
        class a{
            virtual int fun() = 0;
        };
        class b : public a{
            int fun() override;
        };
        class c : public b{
            int fun() override;
        };

// * Keywords "virtual" and "override" are not addedd in the .cpp file.

// * If we want a quicker access to a value during the compilation:
        constexpr static double val = 0.0;  // "0.0" will be directly substituted to val everywhere

// * The initialization of a static member is performed in the .cpp file:
        class Example{               // Example.h
            static int static_member;
        };
        int Example::static_member = 0;  // Example.cpp
```

```cpp
// * If we return a typedef of a class, we must specify from which class we take it in the .cpp:
        class Example{                                          // Example.h
            typedef std::vector<int> vec_of_int;
            vec_of_int function(..);
        };
        Example::vec_of_int Example::function(..){ .. }  // Example.cpp

// * If we have an associative container and a const method, we cannot do (for example):
        return element[position];
//   because this procedure creates an element with key=position if the element doesn't exists.
//   Instead we should do:
        return element.at(position);

//------------------------------------------
// POINTERS and REFERENCES
//------------------------------------------
// * Functions call with pointers and references:
        void fun(int* r);   // we expect an address: fun(p)/fun(&x); (int x, int* p)
        void fun(int& r);   // we expect an integer: fun(x); (int x)

// * Arrays are passed as pointers, not as a copy: to guarantee a function won't change them -> const:
        int arr[100];
        void fun(const int arr[]);

// * We pass strings as references (they might be large):
        void fun(std::string& s);

// * To pass a matrix (array of arrays) we have to pass the second dimension:
        int matrix[10][10];
        void fun(const int matrix[][10]);

// * MATRIX: array of arrays: we can extract rows (not columns)
        int matrix[10][10];
        matrix[0];              // 1st row

// * When we define iterators it is more conveniente to use "auto".

// * If we want a class wich elements share something:
        class Events{ .. };
        class Calendar{
            std::shared_ptr<std::vector<Events>> events;
            public:
                Calendar() : events(std::make_shared<std::vector<Events>>()){}
        };
//   How to access events? If we would have had some member function that would have used the vector events
//   and we would have wanted an iterator to the vector of events or a for-loop, we would have written:
        std::vector<Events>::iterator it = events->begin();
        for(const Event& e : *events){ .. }

// * ATTENTION when we mix pointers and classes:
//      - pointer->member               both data member and functions
//      - (*pointer).member             both data member and functions
//   this behavior is valid both for raw pointers and for shared pointers.

//------------------------------------------
// MPI
//------------------------------------------
// * When ALL ranks has to do something and send it to rank_0, REMEMBER that also is one of those ranks!

// * (!) If we use any sharing-operation we need all the ranks to prepare the containers:
        if(rank==0){
            int var;
            // do something with var (even cin>>var);
            MPI_Send(&var, ..)
        }
        else{
            int var;
            MPI_Recv(&var, ..)
        }

// * Every rank needs something initialized differently?
        #include<random>
        unsigned seed = rank * a_local_variable;
        std::default_random_engine generator(seed);
        std::uniform_real_distribution<double> distribution(a,b);  // U([a,b])
        double random_value = distribution(generator);   // pick from the distribution

// * (!) If rank_0 takes as input a string (or vector) and has to pass it to all, it first has to pass the lenght:
        std::string the_string;                 // default initialized in all ranks
        if(rank==0)
            cin >> the_string;                  // for rank!=0 is still default initalized
        unsigned length = the_string.size();    // for rank!=0 this is 0
        if(rank==0){
            MPI_Bcast(&length, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
            MPI_Bcast(&the_string[0], length, MPI_CHAR, 0, MPI_COMM_WORLD);
        }
        else{
            MPI_Bcast(&length, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
            the_string.resize(length);
            MPI_Bcast(&the_string[0], length, MPI_CHAR, 0, MPI_COMM_WORLD);
        }
```

```
// * (!) VECTORS: mpi::read_vector and mpi::print_vector
        std::vector<double> x = mpi::read_vector(n, "x", MPI_COMM_WORLD);
        std::vector<double> y = mpi::read_vector(n, "y", MPI_COMM_WORLD);
        std::vector<double> z = x+y;
        mpi::print_vector(z, n, "Result vector z:", MPI_COMM_WORLD);
//   The first call (mpi::read_vector()) takes as input the vector x (through cin) and divides the vector
//   in many (#ranks) local vectors x (through MPI_Scatter()). After this call every rank will have its
//   local part stored in x. Same goes for the vector y. Notice that the input to the call will be the
//   whole vector x, the function performs the splitting automatically.
//   Then all ranks sum the local vectors x and y and store the result in the local vector z.
//   The call mpi::print_vector() builds a signle vector from locals (through MPI_Gather()) and rank_0 prints it.

// * MPI_Reduce():
//   MPI_Reduce(const void *sendbuf, void *recvbuf, .., int dest, MPI_Comm comm);
//   MPI_Reduce(&var1, &var2, .., k, MPI_COMM_WORLD);
//      - if(rank==rank_k) -> it matters both &var1 and &var2
//      - if(rank!=rank_k) -> it matters only &var1
        if(rank==0)
            MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        if(rank==1)
            MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
//   In this case:    - rank_0 takes care both of sending and receiving
//                    - rank_1 takes care only of sending
//                    - rank_0 sends its a to rank_0's b
//                    - rank_1 sends its c to rank_0's b
//                    - rank_1's "&d" is completely neglected (equivalent to "nullptr")
//   This rule applies always when we have MPI_Call( .. *sendbuf, .. *recvbuf, .. ).

// * (!) When we use a parallel function we need to use the "ALL" version of every sharing-operation!
        double function(..){
            double local_var = .. ;
            // something with local_var;
            double global_var;                  // (!)
            MPI_Allreduce(&local_var, &global_var, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
            return global_var;
                }
//   (!) If we don't procede like this, every rank!=0 (e.g.) will return a local result!

// * If rank_0 needs to pass an nxp matrix:
//      - rank_0 needs to pass n (MPI_Bcast(&n, ..))
//      - rank_0 needs to pass p (MPI_Bcast(&p, ..))
//      - other ranks need to create a local_matrix(n,p); (initialized with zeros)
//      - rank_0 can pass now the matrix:
                MPI_Bcast(&matrix.data(), n*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
//      - other ranks need to store the (whole) matrix:
                MPI_Bcast(&local_matrix.data(), n*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// * If rank_0 wants to split a matrix (M) in sub-matrices:
//      - rank_0 need to pass n (MPI_Bcast(&n, ..))
//      - rank_0 will pass to every rank local_n = n/#processes columns and so
//        every rank has to perform: ln = n/size;
//      - (!) every rank has to construct a local matrix of the new size: loc_M(ln,p);
//      - rank_0 can scatter now the matrix:
                MPI_Scatter(&M.data(), ln*p, MPI_DOUBLE, &loc_M.data(), ln*p, MPI_DOUBLE, 0, COMM_WORLD);

// * MPI_Bcast() and MPI_Reduce/MPI_Allreduce don't need to be re-written/paired in different ranks
//   MPI_Send()    -> MPI_Recv()
//   MPI_Scatter() -> MPI_Gather()

//-------------------------------------------
// RANDOM
//-------------------------------------------
// * Ternary condition: (condition) ? (if_true) : (if_false);
        int sign = value>=0 ? 1 : -1;  // if value>=0 then sign=1, else sign=-1

// * To generate random numbers:
        #include<random>
        std::default_random_engine generator;
        std::uniform_real_distribution<double> distribution(a,b);   // U([a,b])
        double x_guess = distribution(generator);      // pick from the distribution

// * To get "NaN" as a value: (returnable and of needed type)
        #include<limits>
        double err_val = std::numeric_limits<double>::quiet_Nan();

// * Return "not" means: if not (condition) then return 1:
        return not (condition);

// * Library "matrix.": we should we prefer to implement A(i,j) instead of A[i][j]?
//   In the first case we only need:
        int Matrix::operator()(int i, int j);
//   In the second case we need to implement two operators:
        int Matrix::operator[](int j);
        std::vector<int> Matrix::operator[](int i);
//   the first operator access the row i, the second the the element j.

// * Attention to the assignment and the copy (constructors):
        Example_class a = {0,1,2};
//   we're calling the constructor to create a (default initalize), then we call
//   the copy constructor to change the values. If the constructor has a count++
//   and also the copy constructor has a count++, the count will be "++" twice.
//   In a similar case:
        vector<Example_class> vec = {a, b};
```

```
//   also here we first initialize the elems in the vector -> count++ (constructor)
//   and then we copy the elements a and b -> count++ (copy constructor).

// * The ++ operator in a container:
        vector<int> vec = .. ;
        int index = 0;
        vec[index++] = 4;         // equivalent to: vec[index]=4; index++;

// * ATTENTION at the return of a function:
        vector<int> function(..){
            if(error)
                return vector<int>();
        }

// * (!) MAPS: when we define a map, the value part MUST have the DEFAULT CONSTRUCTOR
        class Example{
            public:
                int value;
                Example() = default;
                Example(int val): value(val){}
        };
        // main
        std::map<int, Example> the_map;
        Example e(1);
        the_map[1] = e;
//   the_map[1] creates a pair with key=1 and value default initialized, then we perform
//   the assignment operation. We need the default constructor explicitly defined.

// * STRING: function "find":
        std::string s = .. ;
        if(s.find(word) != s::npos) { .. }
//   the method .find(word) returns the index of the first character of word inside the string
//   if the word is found, otherwise it return an invalid index ("npos").

// * FUNCTIONAL
        void fun(const std::function<double (const nd_vector&)>& f){ .. }
//   whatever is the function f we can evaluate it (we can do f(elem)).
// * QUEUE: if we want a container with "push" and "pop" functions we use a queue:
//   ("push" adds an element at the end, "pop" returns the first element and then removes it)
        std::queue<int> example;
        example.push(1);
        example.pop();

// * We can use multiple containers if this reduces the complexity (even at the cost of memory).

// * If we have an associative container with value as contianer:
        unordered_map<string, unordered_map<int, PersonalClass>> example;
//   we can do:
        // access a method or a member function of PersonalClass
        example.at(s).at(0).method_of_PersonalClass();
        example.at(s).at(0).public_member_of_PersonalClass = 4;
        example.at(s)[0].method_of_PersonalClass();
        example.at(s)[0].public_member_of_PersonalClass = 4;
        example[s][0].method_of_PersonalClass();
        example[s][0].public_member_of_PersonalClass = 4;
        // inserting a pair given the key s
        example.at(s).insert(pair<int, PersonalClass>(2, PersonalClass(0,0)));

// * Optimization of the containers:
//   to decide we have to analyze all the methods/functions on by one:
//       - need fast access to a book through its title?
                unordered_map<string, Book>
//         generally: fast access through a key -> unordered_map
//       - need fast access to a magazine and its issue_number?
                unordered_map<string, unordered_map<int, Magazine>>
//       - need to fast comparison of elements?
                set<Elements>
//       - need to keep track of the order of insertion?
                map<int, Elements>

// * When we operate with std::pair:
        std::pair<double, double> function(..){
            double d1;
            double d2;
            // something with d1, d2;
            return std::make_pair(d1,d2);
            return {d1, d2};                // alternative
        }
```